UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Algoritmos de ordenación en GPU con OpenCL

Autor: Francisco Alcudia Diaz
Tutor: Iván González Martínez

junio 2021

Francisco Alcudia Diaz
Algoritmos de ordenación en GPU con OpenCL

Francisco Alcudia Diaz

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Este trabajo de fin de grado consiste en el estudio de la computación de propósito general en unidades de procesamiento gráfico y su aplicación práctica sobre el problema de ordenación utilizando Open Computing Language (OpenCL).

En la primera mitad del documento se estudian los paradigmas de computación paralela y computación heterogénea, la arquitectura de GPU Fermi, los frameworks CUDA, OpenCL y SYCL, el estado del arte de los algoritmos de ordenación paralelos para GPUs y los algoritmos Enumeration sort, Merge sort y Radix sort.

En la segunda mitad del documento se describe el trabajo práctico realizado y los resultados. El trabajo práctico consiste en la implementación de los algoritmos Enumeration sort, Merge sort y Radix sort utilizando PyOpenCL, además de la realización de diferentes pruebas sobre éstos y la creación de diferentes versiones para experimentar con diferentes alternativas, parámetros y estrategias.

Palabras clave

GPGPU, computación paralela, computación heterogénea, arquitectura GPU Fermi, OpenCL, algoritmos de ordenación, Enumeration sort, Merge sort, Radix sort

Abstract

This end of degree work consists of the study of general purpose computing on graphics processing units and its practical application on the sorting problem using Open Computing Language (OpenCL).

In the first half of the document, different theoretical topics have been studied. These are the paradigms of parallel computing and heterogeneous computing, the GPU architecture Fermi, the heterogeneous computing frameworks CUDA, OpenCL and SYCL, the state of the art of parallel sorting algorithms for GPUs and the sorting algorithms Enumeration sort, Merge sort and Radix sort.

The second half of the document contains the description of the practical work carried out and the results. The practical work consists of the implementation of the algorithms Enumeration sort, Merge sort and Radix sort using PyOpenCL, in addition to carrying out different tests on them and creating different versions to experiment with different alternatives, parameters and strategies.

Keywords

GPGPU, parallel computing, heterogeneous computing, GPU architecture Fermi, OpenCL, sorting algorithms, Enumeration sort, Merge sort, Radix sort

ÍNDICE

LISTAS

Lista de figuras

1

INTRODUCCIÓN

1.1. Contexto

En el pasado, los fabricantes de microprocesadores mejoraban el rendimiento de éstos a un ritmo constante, observado y predicho por Gordon Earl Moore, esta predicción se conoce comúnmente como «La ley de Moore». Este ritmo constante de mejora se mantenía, principalmente, debido al aprovechamiento de las ventajas que se obtenían al reducir el tamaño de los transistores que formaban los microprocesadores. El ritmo de mejora se mantuvo del mismo modo hasta la década de los años 2000.

En la década de los años 2000, los fabricantes de microprocesadores comenzaron a encontrarse con nuevos problemas debidos al tamaño alcanzado de los transistores. Para mantener el ritmo de mejora, decidieron diseñar microprocesadores con más de un núcleo. Este hecho introdujo el paradigma de computación paralela.

Posteriormente, con el objetivo de resolver problemas computacionales más rápido, se comenzaron a utilizar sistemas de cómputo heterogéneos, principalmente formados por una CPU y una GPU, donde se emplean modelos de computación heterogénea.

La aparición de estos paradigmas de computación y sistemas con mayores capacidades permite y permitió mejorar el rendimiento de software antiguo al adaptarlo a nuevas tecnologías, y la creación de software nuevo que resuelve los mismos problemas con mayor rendimiento y eficiencia [1].

Uno de los problemas fundamentales de la computación es el problema de ordenación u ordenamiento, los algoritmos que resuelven estos problemas son muy útiles para optimizar las soluciones a otros problemas como, principalmente, los problemas de búsqueda. El problema de ordenación ha sido estudiado durante mucho tiempo y ha recibido muchas soluciones distintas siguiendo modelos de computación tradicionales.

En la última década, el problema ha vuelto a ser estudiado y ha recibido nuevas soluciones diseñadas principalmente para GPUs. Estas nuevas soluciones se han desarrollado con nuevas tecnologías que emplean modelos de computación paralela y heterogénea [2] [3] [4].

1.2. Motivación

La motivación para realizar este trabajo de fin de grado es mi interés por aprender sobre computación en sistemas heterogéneos, principalmente los formados por una CPU y una GPU, y por ampliar mis conocimientos sobre algoritmos de ordenación.

Puesto que en el punto de partida mis conocimientos son nulos sobre modelos de computación heterogénea y GPUs, este trabajo se centrará más en dichos conceptos y se dedicará menos tiempo a los algoritmos de ordenación en sí, los cuales ya son estudiados durante el grado.

1.3. Objetivos

Este trabajo de fin de grado tiene como objetivo principal servir de introducción al mundo de la computación heterogénea, y más concretamente, al mundo de la computación de propósito general en unidades de procesamiento gráfico. Es por ello que el trabajo se enfocará en el tipo de sistema heterogéneo formado por una unidad central de procesamiento y una unidad de procesamiento gráfico, el cual es uno de los tipos de sistemas heterogéneos más comunes que encontramos en ordenadores personales. El trabajo se plantea como un trabajo con enfoque científico.

Se pretende, como trabajo teórico y en primer lugar, estudiar los paradigmas de computación paralela y computación heterogénea, analizar la arquitectura de una GPU relativamente moderna y revisar las herramientas disponibles que ofrecen modelos de programación compatibles con los tipos de computación nombrados.

En segundo lugar se pretende revisar el estado del arte de algoritmos de ordenación paralelos orientados a GPUs y posteriormente seleccionar algunos de éstos con los que trabajar en la parte práctica del trabajo. No se pretende mejorar ni competir con las implementaciones existentes, se pretende conocer cuales son los algoritmos que mejor se adaptan a las arquitecturas GPUs y cuales son las técnicas que emplean sus autores.

Por otro lado, como trabajo práctico, se pretende, en primer lugar, experimentar con los conceptos y detalles tratados en la parte teórica e implementar los algoritmos seleccionados haciendo uso de una de las tecnologías estudiadas. Se utilizarán estas implementaciones para recolectar métricas relacionadas con el rendimiento.

Por último, se reunirán los datos obtenidos y se realizarán comparaciones entre los algoritmos, las diferentes estrategias planteadas para implementarlos, y el rendimiento de distinto hardware sobre el que se probarán las implementaciones.

1.4. Estructura del documento

El cuerpo de este trabajo de fin de grado está compuesto por los siguientes capítulos:

1.– Introducción

El primer capítulo da un contexto al trabajo, expone la motivación por la cual se decidió elegir el tema del trabajo, describe los objetivos que se pretenden conseguir y describe los capítulos que forman el documento.

2.– Estado del arte

Trata los conceptos de computación paralela, computación heterogénea y computación de propósito general en GPUs. Describe una arquitectura GPU, varios frameworks que permiten crear aplicaciones para sistemas heterogéneos y los algoritmos de ordenación con los que se trabaja en la parte práctica.

3.– Desarrollo

Describe como se han implementado los distintos algoritmos de ordenación y como se han recolectado los datos con los que posteriormente se han creado gráficas.

4.– Experimentos

Muestra y discute los resultados parciales y las pruebas realizadas durante el desarrollo.

5.– Resultados

Extiende los resultados parciales con los resultados de otros sistemas y discute los diferentes resultados obtenidos.

6.– Conclusiones y trabajo futuro

Contiene las conclusiones alcanzadas tras realizar el trabajo y las líneas en las que se podría seguir trabajando.

2

Estado del arte

2.1. Computación paralela

La computación paralela es una forma de computación en la que se lleva a cabo más de una operación al mismo tiempo, basándose en la idea de que los problemas pueden dividirse en problemas más pequeños y ser resueltos de forma concurrente. Existen dos tipos de paralelismo en relación con los problemas computacionales, éstos son paralelismo a nivel de dato y paralelismo a nivel de tarea.

El paralelismo a nivel de dato consiste en la distribución de los datos entre los recursos computacionales, de modo que cada recurso computacional trabaja únicamente con los datos que se le asignan y obtiene un resultado parcial del problema, estos resultados parciales del problema son posteriormente recogidos y unificados con el fin de obtener el resultado global. Generalmente se aplica la misma operación sobre todos los datos.

El paralelismo a nivel de tarea consiste en distribuir diferentes tareas que realizan operaciones sobre los mismos o distintos grupos de datos entre los recursos computacionales disponibles. Las tareas deben ser independientes entre ellas para poder ser realizadas de forma concurrente.

El nivel de paralelismo que puede alcanzar un algoritmo depende de las características del problema que se pretende resolver. Siempre debe existir cierta independencia entre sus operaciones, datos o tareas para que sea posible diseñar un algoritmo bajo el modelo de computación paralela. El modelo de computación paralela introduce nuevos problemas a tener en cuenta que no existen en la computación no paralela, éstos están relacionados con la compartición de memoria y la sincronización de recursos.

Las operaciones en distintos recursos computacionales pueden requerir acceso a las mismas posiciones de memoria para realizar operaciones de lectura o escritura por lo que son necesarios modelos de memoria consistentes que tengan en cuenta los problemas que pueden surgir. Por otro lado, puede ser necesario llevar a cabo la sincronización del estado de las tareas en ciertos problemas. Para ello, de igual manera, se requiere de modelos y mecanismos que permitan llevar a cabo la sincronización entre tareas de forma correcta [1].

2.2. Computación heterogénea

La computación heterogénea es el paradigma que se emplea para desarrollar programas que se ejecutan en sistemas heterogéneos. Éstos son sistemas formados por unidades de procesamiento con distinta arquitectura. La razón principal por la que existen este tipo de sistemas es por el hecho de que cada arquitectura se adapta mejor a ciertos tipos de problemas y, por tanto, es posible tener sistemas con un mayor rendimiento general.

Este modelo de computación se enfrenta a más dificultades con respecto a un modelo de computación homogénea. Requieren, por ejemplo, de modelos de ejecución y comunicación entre las distintas unidades de procesamiento que forman el sistema. Por otro lado, la existencia de gran cantidad de arquitecturas y fabricantes de unidades de procesamiento tiene como consecuencia la existencia de un gran número de entornos y soluciones no portables de unas plataformas a otras, por lo que conviene tener un estándar de computación heterogénea.

Existen numerosos tipos de sistemas heterogéneos, el más común de ellos está formado por una unidad central de procesamiento (CPU), la unidad computacional que normalmente se encarga de organizar el sistema, y uno o más coprocesadores. Como coprocesadores podemos encontrar unidades de procesamiento gráfico (GPU), procesadores de señal digital (DSP), FPGAs, unidades de procesamiento neuronal (NPU), etc. Algunos de estos coprocesadores están diseñados para solucionar un problema en concreto, pero en algunos casos pueden ser utilizados para realizar computación de propósito general, como en el caso de las GPUs [1].

2.3. GPGPU

Inicialmente, la función principal de las unidades de procesamiento gráfico era, como su nombre indica, el procesamiento de gráficos. De esta forma se lograba liberar a la unidad central de procesamiento de las grandes cargas de trabajo que requieren las aplicaciones relacionadas con imágenes o vídeo, como los videojuegos.

En la actualidad, es posible aprovechar la potencia de las unidades de procesamiento gráfico en aplicaciones no relacionadas con gráficos, es decir, es posible desarrollar algoritmos que no trabajan con gráficos y ejecutarlos en unidades de procesamiento gráfico. Esto se conoce comúnmente como computación de propósito general en unidades de procesamiento gráfico o GPGPU, por sus siglas en inglés. La principal característica de las GPUs es que son dispositivos con un gran nivel de paralelismo, por lo que, para aprovechar la arquitectura de éstas y acelerar un algoritmo, se requiere que el problema o algoritmo sea paralelizable [1].

2.4. Arquitectura de una GPU

En este capítulo se resume brevemente la historia de las GPUs y se analiza la arquitectura Fermi de Nvidia. Se ha elegido analizar esta arquitectura porque la GPU que se utilizará para la parte práctica es de una arquitectura muy similar. Este capítulo está basado en [5].

2.4.1. Historia

La historia de las GPUs empieza en la década de 1980 con los aceleradores de gráficos 3D no programables, estos dispositivos eran motores de renderizado 3D que se empleaban principalmente para programas relacionados con gráficos.

Fue en 2001 cuando Nvidia introdujo en estos dispositivos la posibilidad de programar una parte de ellos, fue entonces cuando la computación de propósito general en GPUs comenzó a surgir. Por aquel entonces, estos dispositivos eran muy poco flexibles, estaban pensados únicamente para realizar operaciones relacionadas con gráficos 3D, y esto se reflejaba en el diseño de sus arquitecturas.

Éstas eran segmentadas, por etapas, donde en cada una de ellas se realizaba un tipo de computación especifica ligada al procesamiento de gráficos 3D y para utilizarlas para computación de propósito general había que trabajar como si se estuviese trabajando con gráficos 3D.

Fue a mediados de la década de los años 2000 cuando comenzaron a surgir arquitecturas que eliminaban esas etapas, donde se realizaban cálculos muy específicos, unificándolas en una sola capaz de sustituir a las anteriores. Esto facilitó la tarea de realizar computación de propósito general en GPUs y fue acompañado de la aparición de herramientas que ofrecían un modelo de programación más fácil y accesible, es decir, los diseños de las arquitecturas de GPUs comenzaron a desarrollarse teniendo en cuenta los beneficios que podía generar el hecho de que éstas pudiesen ser utilizadas para realizar computación de propósito general.

2.4.2. Arquitectura Fermi

La arquitectura de GPU que se describe a continuación es la arquitectura Fermi de Nvidia, las GPUs con esta arquitectura salieron al mercado en abril de 2010. Un chip desarrollado bajo esta arquitectura, cuyo diseño a alto nivel se muestra en la figura 2.1, consiste en un número variable de los denominados Streaming Multiprocessors o SMs, cuyo diseño a alto nivel se muestra en la figura 2.2. La tecnología equivalente a estos SMs, en AMD, son las llamadas Compute Units o CUs.

En la figura 2.1 nos encontramos con 16 SMs representados por rectángulos verticales que contienen otros rectángulos de distintos colores, los cuales guardan relación con los colores de los componentes que forman un SM en la figura 2.2.

Además de los SMs, se observa en la figura 2.1 que el chip está compuesto por múltiples interfaces de memoria RAM, una interfaz para el anfitrión, una caché de nivel dos situada en el centro del chip y un componente llamado GigaThread.

Cada uno de los Streaming Multiprocessors está formado por 32 núcleos o procesadores escalares, los cuales son llamados CUDA cores por Nvidia, son capaces de ejecutar una operación de enteros o coma flotante por ciclo de reloj. Estos 32 CUDA cores están divididos en dos unidades de ejecución como se observa en la figura 2.2, cada una de estas dos unidades contiene 16 procesadores escalares.

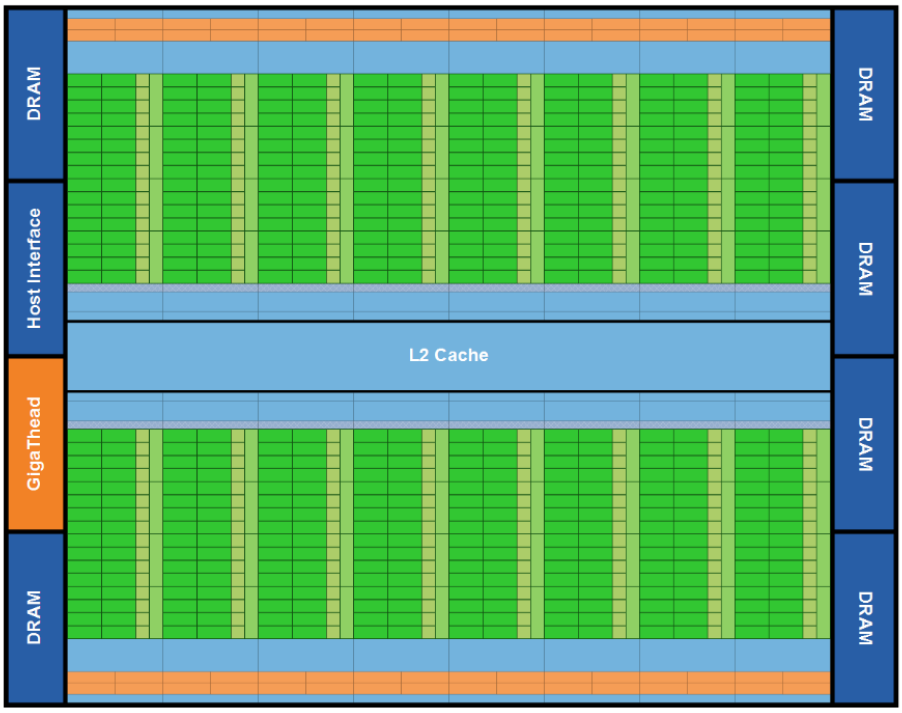

Figura 2.1: Esquemático del diseño a alto nivel de un chip con arquitectura Fermi [5]

Cada uno de los SMs, además de los 32 CUDA cores, contiene una caché de instrucciones, dos unidades planificadoras de warps, dos unidades para cargar las instrucciones, 4096 registros con un tamaño de palabra de 32 bits, 16 unidades para realizar operaciones de escritura y lectura y cuatro unidades para llevar a cabo funciones especiales como la función seno y la función coseno.

Otro de los componentes que contiene cada SM es una memoria de 64 kB compartida por todos los componentes del mismo SM, la cual también se utiliza como caché de primer nivel. Esta memoria es configurable, permite decidir cual es la cantidad de memoria que se utiliza como memoria compartida y cual es la cantidad que se utiliza como caché de primer nivel. Puede configurarse en dos posibles modos.

Uno de los modos permite utilizar 48 kB como memoria compartida y los 16 kB restantes como memoria caché de primer nivel. Este tipo de configuración es más indicado para programas que utilicen mucha memoria compartida.

El otro modo permite utilizar 48 kB de la memoria como caché de primer nivel y 16 kB como memoria compartida. Este caso es mejor para programas en los que no se conocen de antemano las posiciones de memoria que se utilizarán.

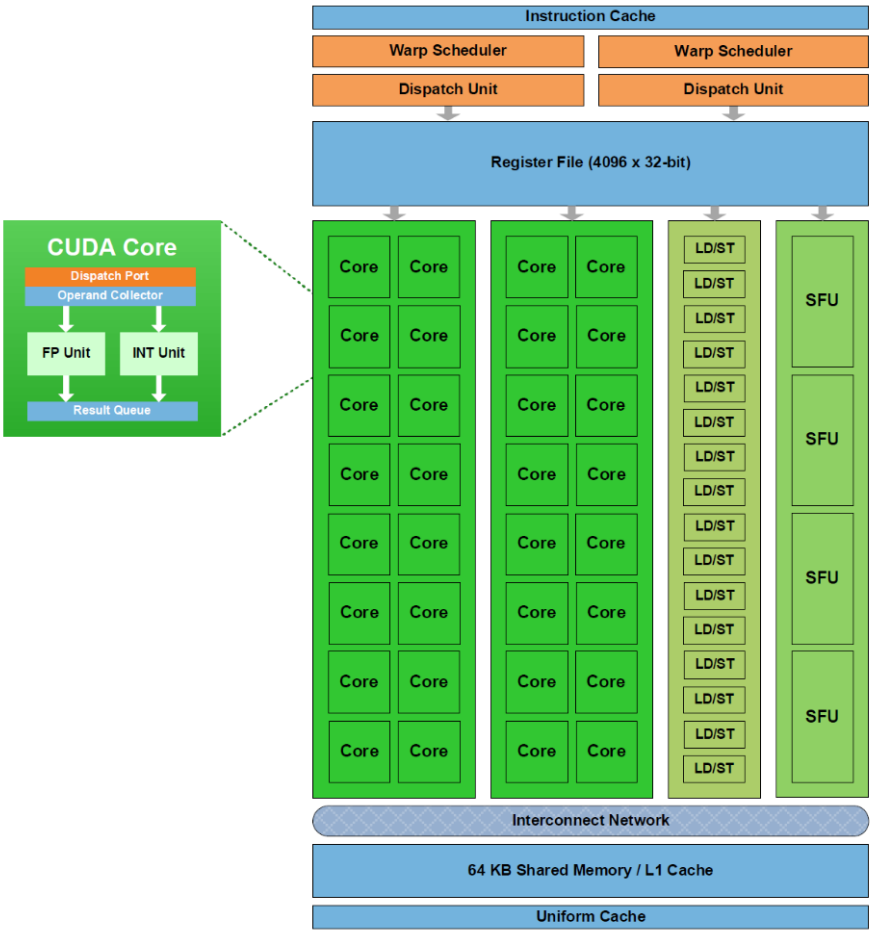

Figura 2.2: Esquemático del diseño a alto nivel de un SM de la arquitectura Fermi [5]

De vuelta al funcionamiento general del chip, el componente llamado GigaThread es básicamente un panificador que se encarga de distribuir la carga de trabajo entre los distintos SMs. Los programas que se ejecutan en GPUs están compuestos por uno o más kernels. Un kernel se puede definir como una función. Cada kernel está formado por un número determinado de hilos que se ejecutan en paralelo, los cuales se agrupan en grupos de hilos.

Cada grupo de hilos se ejecuta en un único Streaming Multiprocessor, de forma que aquellos hilos que pertenecen al mismo grupo, y por tanto se ejecutan en el mismo SM, pueden compartir los recursos del SM, como la memoria local, anteriormente referida como memoria compartida o caché de primer nivel.

Todos los hilos de un mismo grupo pueden sincronizarse para realizar operaciones de memoria sobre la memoria global, al igual que pueden sincronizarse para realizar operaciones de memoria sobre la memoria local.

Los grupos de hilos, a su vez, se dividen en grupos de 32 hilos para ser ejecutados al mismo tiempo, los cuales se conocen como warps. Un warp es la unidad mínima que puede ser cargada para ejecutarse en un Streaming Multiprocessor.

En la arquitectura Fermi, como se observa en la figura 2.2, cada SM tiene dos unidades de planificación de warps. Esto se debe a que en esta arquitectura se pueden cargar 32 instrucciones de un mismo warp o 16 instrucciones de dos warps distintos en un mismo ciclo de reloj, siempre y cuando los recursos del SM lo permitan. Esto quiere decir que varios grupos de hilos pueden ejecutarse al mismo tiempo en un mismo SM si hay recursos suficientes para ambos.

Esto es posible porque, como se describe anteriormente, los 32 procesadores escalares están divididos en dos grupos de ejecución. Cada uno de estos grupos de ejecución es una unidad SIMD (Single Instruction, Multiple Data). Esto quiere decir que cada uno de los 16 procesadores lleva a cabo la misma instrucción sobre distintos datos. Por lo que cada SM es una unidad MIMD (Multiple Instruction, Multiple Data).

Por otro lado, los grupos de hilos forman kernels como se ha visto anteriormente. Y esta arquitectura permite ejecutar más de un kernel al mismo tiempo, siempre que estos formen parte del mismo programa. Si la misma GPU que se utiliza para realizar GPGPU se está utilizando como salida de vídeo del sistema, la pantalla se congelará cuando se ejecuten aplicaciones de propósito general porque la aplicación que refresca la pantalla no entrará en ejecución.

2.5. Frameworks para computación heterogénea

En este capítulo se presentan algunos frameworks que permiten desarrollar aplicaciones para sistemas heterogéneos. Éstos se describen de manera breve, excepto OpenCL, que se utilizará en la parte práctica de este trabajo.

2.5.1. CUDA

CUDA es un framework desarrollado por Nvidia el cual se define como un modelo de programación y plataforma de computación paralela para realizar computación de propósito general en unidades de procesamiento gráfico [6].

La principal desventaja que tiene este framework es que las soluciones creadas con este entorno de trabajo no son directamente portables a otras plataformas, únicamente pueden ejecutarse en GPUs de Nvidia. Si se quisiese utilizar una solución en una GPU de AMD sería necesario volver a implementar la solución con otra tecnología o utilizar alguna herramienta que transformase la solución a otra tecnología. Existen otros frameworks, como OpenCL, para evitar esta dependencia a un único fabricante.

Por otro lado, la principal ventaja de CUDA es su rendimiento y librerías. Como se demuestra en [7], CUDA tiene un rendimiento superior a OpenCL, o al menos lo tenía en 2011, y se recomienda elegirlo frente a OpenCL cuando el rendimiento de las aplicaciones sea muy importante. En el caso de este trabajo el rendimiento no es tan importante, por lo que no se utilizará CUDA, se utilizará OpenCL puesto que así será posible ejecutar las soluciones en muchos más dispositivos.

2.5.2. OpenCL

El framework Open Computing Language, más conocido como OpenCL, es una tecnología estándar para el desarrollo de aplicaciones para sistemas heterogéneos, aunque también permite desarrollar aplicaciones para sistemas homogéneos. Está desarrollado por el consorcio Khronos Group.

Este framework está compuesto por una interfaz de programación, que permite al desarrollador abstraerse de los diferentes tipos de plataformas y vendedores hardware, y el lenguaje OpenCL C, que es el lenguaje que se debe utilizar para programar las aplicaciones que se quieren ejecutar en los dispositivos de cómputo. Cada fabricante o vendedor de dispositivos de cómputo debe desarrollar su propia implementación de OpenCL para que sus dispositivos sean compatibles con esta tecnología.

La arquitectura y funcionamiento de OpenCL se describe mediante cuatro modelos: un modelo de plataforma, un modelo de ejecución, un modelo de programación y un modelo de memoria [1] [8].

Modelo de plataforma

El modelo de plataforma de OpenCL define que una plataforma consiste en un anfitrión y uno o más dispositivos de cómputo compatibles con OpenCL. El anfitrión es, generalmente, una CPU, y los dispositivos compatibles con OpenCL son coprocesadores como GPUs, FPGAs e incluso la misma CPU. La CPU del mismo equipo puede ser el anfitrión y el coprocesador al mismo tiempo, por lo que OpenCL también permite desarrollar aplicaciones para sistemas homogéneos.

Las especificaciones de OpenCL definen que los dispositivos compatibles con OpenCL se dividen en unidades de cómputo, las cuales, a su vez, están divididas en elementos de procesamiento. Son estos elementos de procesamiento los que realizarán las tareas computacionales.

El papel principal del anfitrión es organizar y orquestar la ejecución de la aplicación. Para ello, la aplicación envía comandos desde el anfitrión a los distintos dispositivos de cómputo para indicarles que deben hacer. Por lo que el papel de los dispositivos de cómputo es obedecer al anfitrión. Los dispositivos de cómputo realizan comandos de transferencia de datos y de ejecución de kernels principalmente.

Modelo de ejecución

El modelo de ejecución define que las aplicaciones OpenCL están formadas por dos partes: el programa que ejecuta el anfitrión y los kernels, que se definen como funciones declaradas en un programa OpenCL y ejecutadas por un dispositivo de cómputo compatible con OpenCL.

Cada instancia de un kernel recibe el nombre de work-item y cada uno de los work-items recibe un identificador accesible desde el código de los kernels. Esto es útil para que cada instancia sepa sobre que datos tiene que actuar. Los work-items son ejecutados en elementos de procesamiento.

Los work-items son agrupados en work-groups, que al igual que los work-items, reciben un identificador que permite a los work-items saber en que work-group están. Cada uno de estos work-groups es ejecutado en una única unidad de cómputo.

Para poder ejecutar kernels, el anfitrión debe crear un contexto en una plataforma. Seguidamente debe crear una cola de comandos dentro de dicho contexto y asociarla a los dispositivos de cómputo de la plataforma que se desean utilizar. El anfitrión puede solicitar a los dispositivos de cómputo que ejecuten kernels a través de estas colas de comandos.

A través de estas colas de comandos el anfitrión puede enviar comandos de tres tipos, comandos de ejecución de kernels, comandos relacionados con transferencias de memoria y comandos para sincronizar la ejecución de kernels.

El código fuente de los kernels debe ser cargado por el código del anfitrión en tiempo de ejecución. Si se trabaja con código fuente, éste es compilado en tiempo de ejecución antes de ser enviado a los dispositivos de cómputo. También se puede trabajar con ficheros binarios previamente compilados del mismo modo pero que, igualmente, deben ser cargados por el anfitrión.

Modelo de programación

El modelo de programación no impone un modelo de programación paralela estricto, permite al desarrollador decidir el número de work-items que se va instanciar para cada kernel y el tamaño de cada uno de los work-groups en los que se van a agrupar los work-items. El tamaño de cada uno de los work-groups no es necesario que lo seleccione el desarrollador, puede dejarse a elección de la plataforma que se esté utilizando.

Este modelo de programación define que únicamente es posible sincronizar work-items del mismo work-group, es decir, aquellos work-items que se ejecuten en la misma unidad de cómputo. No define ningún mecanismo para sincronizar work-items de distintos work-groups. Aunque esto puede lograrse mediante la sincronización de kernels en una cola de comandos.

Puesto que el modelo permite sincronizar colas de comandos, es posible enviar dos comandos de ejecución del mismo kernel y esperar hasta que ambos hayan finalizado. Una vez hayan finalizado, se

pueden volver a lanzar los mismos kernels.

Por último, el modelo de programación define objetos de memoria llamados buffers para gestionar la memoria entre el anfitrión y los dispositivos de cómputo.

Modelo de memoria

El modelo de memoria de OpenCL clasifica la memoria que usan las aplicaciones en cuatro tipos: memoria global, memoria constante, memoria local y memoria privada. Cada tipo tiene unas restricciones diferentes.

La memoria global es accesible por el anfitrión, quien siempre la reserva y asigna de forma dinámica, el anfitrión tiene acceso de escritura y lectura. Por otro lado, también es accesible para todos los work-items de todos los work-groups. En esta memoria pueden leer y escribir pero no pueden reservarla e inicializarla.

La memoria constante es igual que la memoria global para el anfitrión. Para los work-items es aquella a la que pueden acceder todos, independientemente del work-group al que pertenezcan, pero, a diferencia de la memoria global, en este tipo de memoria los work-items solo tienen capacidad para realizar lecturas y asignarla de forma estática.

La memoria local es la memoria a la que el anfitrión no tiene acceso de lectura ni escritura pero sí puede inicializarla de forma dinámica. Con respecto a los work-items, es aquella a la que pueden acceder todos los work-items de un mismo work-group para realizar operaciones de lectura y escritura. Además pueden reservarla de manera estática.

Por último, la memoria privada es aquella que no es asignable ni accesible por el anfitrión. Este tipo de memoria es accesible por un único work-item, quien, además, puede asignarla de forma estática.

Relación entre arquitectura Fermi y OpenCL

Como se observa, la terminología empleada por Nvidia para darle nombre a los diferentes conceptos y componentes de su arquitectura se asemeja mucho a los conceptos definidos por la especificación de OpenCL. La especificación de OpenCL está basada en la arquitectura de las GPUs.

El mapeo entre la arquitectura Fermi revisada anteriormente y los conceptos definidos por OpenCL se puede intuir. El concepto de dispositivo de cómputo corresponde obviamente con la GPU o chip. Las unidades de cómputo se corresponden con los Streaming Multiprocessor o SMs. Los elementos de procesamiento se corresponden con las unidades de ejecución de tipo SIMD dentro de los SMs.

La memoria global y constante de OpenCL se mapea con la memoria externa a la GPU, la memoria local de OpenCL se corresponde con la memoria compartida dentro de los SMs, y la memoria privada de OpenCL se corresponde con los registros internos de los SMs.

Respecto a los hilos, grupos de hilos, kernels y programas de Nvidia. Los hilos se corresponden a los work-items de OpenCL, los grupos de hilos se corresponden a los work-groups, y los kernels y programas se corresponden a los kernels y programas de OpenCL.

Estructura estándar de una aplicación de OpenCL

El código del anfitrión de la mayoría de aplicaciones de OpenCL puede dividirse en cuatro partes. De las cuales, al menos tres de ellas son muy similares. Podemos dividir el código en las siguientes cuatro fases: inicialización, transferencia de datos hacia el dispositivo de cómputo, ejecución de los kernels y transferencia de datos hacia el anfitrión.

En la fase de inicialización se debe obtener alguna de las plataformas instaladas en el sistema y los dispositivos que pertenezcan a ésta y se quieran utilizar. Seguidamente se debe crear un contexto con los dispositivos que se hayan obtenido para utilizarlos y también se debe crear una cola de comandos en el contexto, para enviar los comandos a los dispositivos. En esta fase también se carga el código o binario del programa que contiene los kernels que se quieren utilizar. En caso de cargar el código, éste se debe compilar.

En la fase de transferencia de datos hacia los dispositivos de cómputo se deben crear los buffers correspondientes y generar los comandos de transferencia de datos que envían los datos necesarios para los kernels hacia los dispositivos de cómputo.

En la fase de ejecución de los kernels se debe configurar cómo se van a ejecutar los kernels y simplemente mandar los comandos correspondientes, esta es la parte del código que menos coincide puesto que está muy relacionada con los kernels que se quieren ejecutar y la forma en que éstos han sido implementados. Por último, cuando los kernels finalizan su ejecución, se deben volver a enviar comandos de transferencia de datos para recuperar el resultado de los kernels y liberar los recursos.

PyOpenCL

PyOpenCL es un paquete de Python que permite utilizar la API completa de OpenCL desde Python, lo que permite aprovechar las ventajas de Python y desarrollar el código que se ejecutará en el sistema anfitrión de manera más rápida y fácil, esto también permite centrarse en el código de los kernels, que es el código que realmente impactará en el rendimiento y en los tiempos de ejecución [9].

Las características principales de este paquete son el manejo de errores automático que incorpora, traduce todos los errores a excepciones de Python, de tal modo que reduce mucho la carga de trabajo. Y por otro lado, la capa base está escrita en C++, por lo que su rendimiento se describe como muy bueno. Además utiliza la licencia MIT y tiene documentación en su página web [10].

2.5.3. SYCL

SYCL se autodefine como una capa de abstracción multiplataforma que permite desarrollar aplicaciones para procesadores o sistemas heterogéneos en código C++ donde el código del anfitrión y el código kernel de los dispositivos se encuentra en el mismo fichero fuente [11], lo cual no es posible llevar a cabo con OpenCL.

Este framework, al igual que OpenCL, surge como un estándar y está desarrollada por el consorcio Khronos Group. Está basado en OpenCL y sigue un modelo de programación superior al de OpenCL. Su objetivo es facilitar y mejorar la productividad de desarrollos de aplicaciones heterogéneas. En principio SYCL estaba muy ligado a OpenCL pero en el pasado reciente se ha independizado de éste y pretende tener soporte con otras tecnologías relacionadas como CUDA y OpenMP.

2.6. Algoritmos de ordenación

Los algoritmos de ordenación son algoritmos que cambian la posición de los elementos de una lista o tabla en función de algún tipo de relación entre éstos. Por lo general, los algoritmos de ordenación trabajan con números y letras como elementos, y como relación entre ellos se suele utilizar el orden numérico y el lexicográfico.

En el pasado el problema de ordenación fue ampliamente estudiado y solucionado con diferentes algoritmos e implementaciones. En el presente y en la última década, debido al surgimiento de la computación de propósito general en unidades de procesamiento gráfico, ha vuelto a ser estudiado y ha recibido nuevas soluciones desarrolladas concretamente para GPUs [2] [3] [4].

En el trabajo [2], publicado en 2009, los autores presentan sus versiones de los algoritmos comúnmente conocidos como Radix sort y Merge sort implementadas con el framework CUDA. La versión de Radix sort presentada era la implementación publicada de un algoritmo de ordenación en GPU más rápida hasta la fecha en que presentaron su trabajo. Y la versión de Merge sort era la implementación publicada de un algoritmo de ordenación basado en comparación más rápida. Para alcanzar el rendimiento al que llegaron indican que descompusieron los problemas en tareas independientes con mínima comunicación global y que aprovecharon la gran velocidad de la memoria local de las unidades de cómputo.

En [3], publicado en 2017, los autores presentan un estudio del estado del arte de algoritmos de ordenación paralelos diseñados para GPUs. En este trabajo se centran en los algoritmos Radix sort, Merge sort, Sample sort y Quick sort puesto que son algunos de los más populares. Concluyen que CUDA-quicksort [12] es uno los mejores algoritmos hasta la fecha.

En [4], publicado en 2017, también se realiza un estudio del estado del arte de algoritmos de

ordenación paralelos para GPUs. Este trabajo recoge una lista de características que reúnen las implementaciones con mejor rendimiento, indican que los algoritmos de ordenación paralelos aprovechan lo máximo posible la memoria local, los hilos deben estar ocupados el máximo tiempo posible, la comunicación y sincronización entre hilos debe realizarse en puntos muy concretos en relación con el hardware que se utilice y la eficiencia con la que se utilicen los registros y la memoria locales son puntos clave. También concluye que las implementaciones basadas en Radix sort son las que consiguen mejores resultados.

Revisando estos trabajos se observa que algunos de los algoritmos más populares son Merge sort y Radix sort, por lo que para la parte práctica de este trabajo se han elegido éstos por su popularidad y rendimiento alcanzable. Y, además, para usarlo en la toma de contacto con OpenCL, se ha elegido Enumeration sort. No es un algoritmo muy eficiente pero sí es muy simple, por eso se elige.

2.6.1. Enumeration sort

Enumeration sort es un algoritmo de ordenación muy sencillo que consiste en calcular la posición final de la lista que debería ocupar cada elemento. Para ello, por cada elemento, se debe contar cuantos elementos deberían ocupar una posición menor en la lista ordenada y posteriormente asignar el elemento a la posición correspondiente.

2.6.2. Merge sort

Este algoritmo es un algoritmo de ordenación basado en comparación. Se le define como un algoritmo de tipo «divide y vencerás». Consiste en dividir la lista de entrada en sublistas hasta llegar a listas de tamaño uno. A partir de entonces, se comienza a unir las sublistas generando listas ordenadas hasta que se unen todas las sublistas. Este algoritmo se basa en dos ideas. La primera idea es que una lista pequeña es más fácil de ordenar que una lista grande, y la segunda es que es más fácil crear una lista ordenada a partir de dos listas ordenadas que a partir de dos desordenadas.

2.6.3. Radix sort

Radix sort no es un algoritmo de ordenación basado en comparación. Este algoritmo ordena números procesando los dígitos de éstos de forma individual. Por cada iteración procesa el dígito de la misma posición de cada elemento y asigna el elemento a una lista correspondiente, de tal manera que en cada iteración se ordenan los elementos por dicho dígito. Cada nueva iteración utiliza como entrada la lista resultante de la iteración anterior. Por lo que tras procesar todos los dígitos, los elementos se encuentran ordenados en la lista resultante de la última iteración.

3

Desarrollo

En primer lugar, se ha decidido que todos los algoritmos de ordenación trabajen únicamente con números enteros de 32 bits y sin signo como claves. Solo se trabajará con claves. Además, se utilizará como relación entre claves el orden numérico, y todos los algoritmos ordenarán de forma ascendente. Cada algoritmo de ordenación debe recibir como parámetro de entrada, al menos, el vector de entrada a ordenar y devolver un vector con los elementos ordenados.

El desarrollo realizado ha consistido, en primer lugar, en implementar el algoritmo Enumeration sort, sobre esta primera implementación se ha estudiado el impacto de los parámetros work-items y work-groups que se utilizan para ejecutar los kernels. Posteriormente se ha implementado otra versión del mismo algoritmo la cual utiliza memoria local con la intención de mejorar el rendimiento.

Tras la implementación de la versión con memoria local del algoritmo Enumeration sort, se ha implementado Merge sort y posteriormente Radix sort. Por último, aprovechando la capacidad de OpenCL de ejecutar el mismo código en distintos tipos de dispositivos, se han probado todas las implementaciones en CPU. Y explorando las limitaciones de OpenCL, se ha implementado una versión de Enumeration sort capaz de utilizar la CPU y la GPU al mismo tiempo, aunque éstas formen parte de plataformas diferentes.

Por otra parte, también se ha desarrollado un sistema para persistir y recolectar los datos de las distintas ejecuciones para posteriormente generar las gráficas que se muestran en los capítulos finales del trabajo. Se ha generado un ejecutable que prueba las implementaciones realizadas y genera un fichero de tipo JSON con los resultados obtenidos. Este ejecutable se ha ejecutado en distintos sistemas para obtener más datos y comparar mi sistema con otros.

3.1. Enumeration sort

La versión inicial de Enumeration sort desarrollada lanza un hilo del kernel por cada elemento en el vector de entrada. Cada uno de estos hilos está asociado a un elemento del vector de entrada, para el que cada hilo calcula cual es su posición final.

Cada hilo itera sobre todos los elementos del vector de entrada, comparándolos con su elemento asociado y contando el número de elementos menores que su elemento asociado. Tras iterar sobre todos los elementos del vector entrada, asignan el elemento asociado a su posición correspondiente.

Por otro lado, la versión que utiliza memoria local para este mismo algoritmo, antes de comenzar a contar cuantos elementos son menores que el elemento asociado, cargan en memoria local un bloque de elementos, procesan ese bloque de elementos y vuelven a cargar otro, de manera sucesiva hasta procesar todos los elementos.

Esta carga de bloques de elementos se lleva a cabo utilizando todos los hilos de cada work-group y sincronizándolos para que carguen los elementos y los procesen a la vez. Esta es una de las técnicas más utilizadas para mejorar el rendimiento de algoritmos en GPUs.

3.2. Merge sort

En la implementación que he realizado de este algoritmo, el kernel se encarga de generar una lista ordenada a partir de dos listas ordenadas del mismo tamaño. El algoritmo está limitado a vectores de entrada con tamaños de potencias de dos.

El código del anfitrión se encarga de calcular los tamaños de las sublistas y el número de hilos del kernel que se lanzan por cada iteración. Con estos parámetros calculados, manda a ejecutar el kernel y, una vez ejecutado el kernel, asigna la salida del kernel a la entrada para utilizar la nueva lista resultante. En cada iteración se lanzan tantos hilos del kernel como parejas de sublistas haya.

Este algoritmo explota la capacidad de paralelización en las primeras iteraciones, pero a medida que avanza, el tiempo de ejecución del kernel aumenta, puesto que cada hilo tiene más datos que procesar y se lanzan menos hilos. No se aprovecha la capacidad de paralelización de las GPUs en las últimas iteraciones.

3.3. Radix sort

Se ha realizado la implementación de la versión binaria de Radix sort. Esta implementación se compone de tres kernels. El primero de ellos genera listas que indican a que grupo pertenece cada elemento. El segundo kernel se utiliza para realizar una suma acumulativa y obtener la posición final de cada elemento. El tercer kernel se utiliza para obtener el vector final utilizando las salidas de los dos kernels anteriores.

El código del anfitrión itera sobre el número de dígitos que tiene cada elemento del vector de entrada, en este caso 32. Siempre realiza 32 iteraciones. Por cada una de estas iteraciones calcula las listas de máscaras lanzando un hilo del primer kernel por cada elemento del vector. Posteriormente se

ejecuta el kernel para llevar a cabo las sumas acumulativas tantas veces como se requiera y por último se obtiene la entrada para la próxima iteración o la lista ordenada en caso de ser la última iteración.

Para la ejecución del primer kernel se lanzan tantos hilos como elementos haya en el vector de entrada. Cada uno de estos hilos asigna a cada elemento un grupo procesando el bit correspondiente a cada ejecución.

El segundo kernel también lanza tantos hilos como elementos haya en el vector de entrada para llevar a cabo la suma acumulativa de elementos en cada grupo. De esta forma se puede calcular de forma paralela la posición que le corresponde a cada elemento en la lista final.

El tercer y último kernel coloca los elementos del vector de entrada en la posición que les corresponde utilizando la salida de los kernels anteriores. Para la ejecución de este kernel también se lanza un hilo por cada elemento del vector de entrada.

Este algoritmo de ha implementado de forma muy sencilla, pero como desventaja, realiza muchos movimientos de memoria entre las ejecuciones de cada kernel. Por otra parte, este algoritmo aprovecha la capacidad de paralelización de las GPUs mucho mejor que el algoritmo Merge sort implementado.

3.4. Enumeration sort CPU y GPU

Este algoritmo adapta la versión inicial de Enumeration sort diseñada para ser ejecutada en un único dispositivo de cómputo para ser ejecutada en dos dispositivos de cómputo. Puesto que OpenCL es un framework de bajo nivel, deja a manos del desarrollador la distribución de la carga de trabajo de una aplicación entre dispositivos de cómputo de distintas plataformas. En mi implementación, el código anfitrión crea un contexto para cada dispositivo de cómputo y compila el código OpenCL para cada plataforma.

A la hora de ordenar el vector de entrada crea los buffers necesarios para cada dispositivo y una cola para cada uno de ellos. La función del código del anfitrión para ordenar los datos recibe como parámetro el porcentaje de la carga de trabajo que se desea que se realice en la CPU o en el primer dispositivo. El código del anfitrión divide la carga de trabajo y manda a ejecutar un primer kernel en ambos dispositivos a la vez, el cual se encarga de generar un vector con el número de elementos menores para cada elemento del vector. Para este primer kernel se ejecuta un hilo por cada elemento del vector de entrada.

Posteriormente, para generar el vector de salida con los elementos ordenados se ejecuta un segundo kernel con un hilo por cada elemento del vector de entrada, mediante el cual se reordenan los elementos del vector de entrada y se obtiene el vector de salida con los elementos ordenados. Para ello utiliza el vector de entrada y el vector de salida del primer kernel que contiene la posición a la que debe asociarse cada elemento. Este kernel se ejecuta en un único dispositivo de cómputo.

Adicionalmente, se ha implementado un mecanismo para obtener la mejor distribución de la carga de trabajo. Simplemente se realiza una prueba con un vector de tamaño pequeño, para ordenar éste se prueban distintas distribuciones para obtener sus tiempos de ejecución y elegir la distribución que logra el menor tiempo.

3.5. Recolección de datos

Para llevar a cabo la recolección de datos, con el objetivo de generar gráficas posteriormente, se ha utilizado TinyDB [13]. Ésta es una base de datos escrita en Python sin dependencias externas. Es una alternativa no relacional a SQLite. Usando esta base de datos se genera un fichero de tipo JSON con los datos recolectados durante la ejecución de los distintos algoritmos de ordenación implementados.

Para probar los algoritmos en más sistemas, se ha utilizado PyInstaller [14]. Esta herramienta genera ejecutables, los cuales contienen el intérprete y las librerías utilizadas de Python, lo que facilita la ejecución en otros equipos. Para generar un ejecutable para Windows, se debe utilizar la herramienta desde Windows, y para generar un ejecutable en Linux, se debe utilizar desde Linux. En mi caso la utilicé desde Windows puesto que mis amigos, quienes me permitieron probar mis algoritmos en sus sistemas y recolectar datos, utilizan Windows.

El proceso fue muy simple, en primer lugar generé el ejecutable con PyInstaller y se lo envié a mis amigos. Ellos simplemente tuvieron que ejecutarlo haciendo doble clic y devolverme el fichero JSON creado. Posteriormente uní todos los ficheros JSON mediante un script de Python utilizando TinyDB y realicé consultas para obtener los distintos datos, generar gráficas, comparar los resultados y discutir la razón de éstos.

4

EXPERIMENTOS

4.1. Plataforma utilizada

Todos los resultados que se muestran en este capítulo pertenecen a las ejecuciones realizadas sobre el mismo sistema. El sistema está formado por una GPU Nvidia GeForce GTX 550 Ti, lanzada al mercado en 2011 por un precio aproximado de 150€. La CPU del sistema es un Intel Core i5-10400F, lanzado al mercado en 2020 por aproximadamente 150€. Sería más interesante tener una CPU y una GPU del mismo año y con un precio de salida similar para poder realizar comparaciones justas.

Los tiempos de ejecución se han medido desde el código Python del anfitrión. Para cada tamaño solo se ha realizado una medición utilizando un vector aleatorio puesto que los tiempos no varían a efectos prácticos para ninguno de los algoritmos implementados en función del estado del vector de entrada.

4.2. Enumeration sort

Como se comenta anteriormente, Enumeration sort se ha utilizado para observar el impacto de algunos parámetros de OpenCL, como el tamaño de los work-groups de un kernel, y número de elementos del vector que procesa cada work-item, lo cual esta relacionado con el parámetro del número de work-items del kernel. Además se ha desarrollado otra versión del algoritmo donde se utiliza una de las técnicas más populares para mejorar el rendimiento en GPUs, ésta consiste en utilizar la memoria local de las unidades de cómputo.

4.2.1. Impacto del tamaño de work-groups

En este apartado se estudia el impacto del tamaño de work-groups utilizado para ejecutar el kernel de Enumeration sort. En este experimento se ordenan 1024 claves, es decir, se lanzan 1024 hilos del kernel probando distintos tamaños de work-groups.

Como se observa en la figura 4.1, cuando el tamaño de los work-groups es el menor posible, es decir 1, se obtiene el peor rendimiento. Esto es así porque al ejecutar los work-groups en los SMs, cada work-group ocupa al menos una unidad de ejecución SIMD de 16 unidades de ancho, y si cada work-group solo contiene un hilo, se están desaprovechando 15/16 por cada unidad de ejecución SIMD. Lo mismo sucede con los valores 2, 4 y 8.

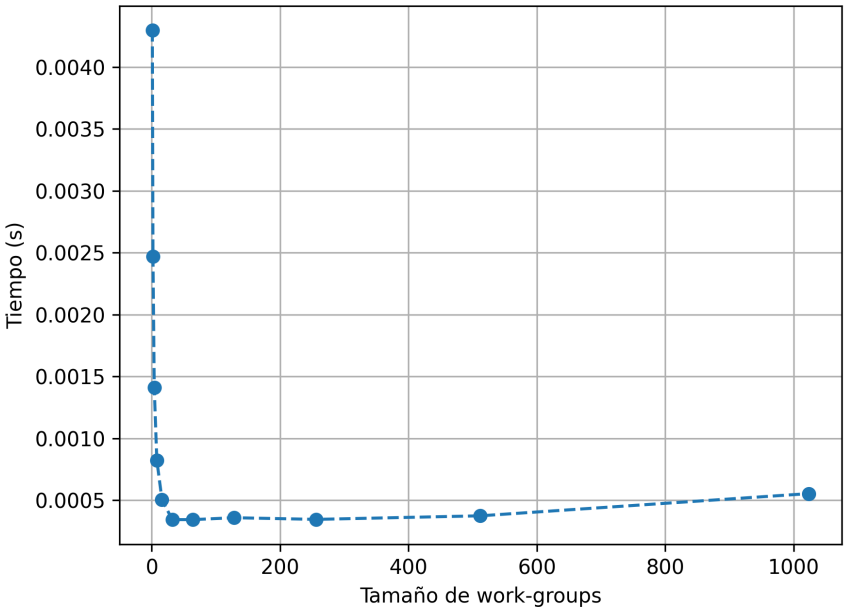

Figura 4.1: Impacto del tamaño de work-groups en Enumeration sort con GTX 550 Ti

Por otro lado, se observa que el rendimiento también empeora cuando el tamaño del work-group es el mayor posible. Esto se debe a que, teniendo que ordenar 1024 claves, y creando un work-group de 1024 hilos, se está forzando a que todos los hilos se ejecuten en el mismo SM, desaprovechando así el resto de SMs.

4.2.2. Impacto del número de elementos a procesar por cada work-item

Este experimento consiste en descubrir cual es la cantidad de elementos óptima que debería procesar cada hilo. Para ello se ha implementado una versión de Enumeration sort en la que cada hilo recibe como parámetro el número de elementos para los cuales debe calcular su posición final.

Como se observa en la figura 4.2, cuando toda la carga de trabajo se realiza en un único work-item, el rendimiento es peor. El rendimiento es mejor cuantos menos elementos del vector de entrada tiene que procesar cada work-item.

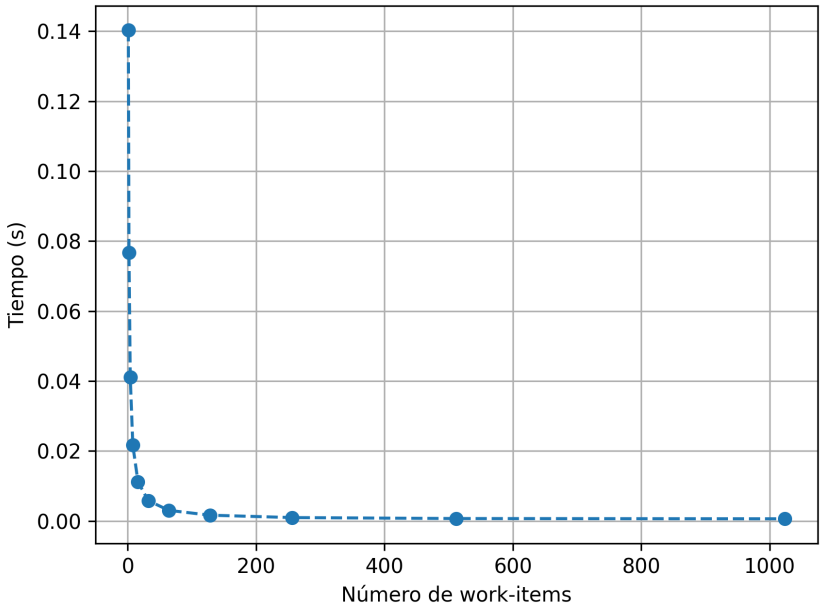

Figura 4.2: Impacto del número de elementos a procesar por cada work-item

4.2.3. Versión con memoria local

En la figura 4.3 se observa la comparación de rendimiento entre la versión inicial que utiliza memoria global de Enumeration sort y la versión posterior que utiliza memoria local. Sorprendentemente el resultado es el contrario al esperado. La versión que utiliza memoria local rinde peor que la que utiliza memoria global. En el capítulo siguiente veremos si sucede lo mismo con otras GPUs.

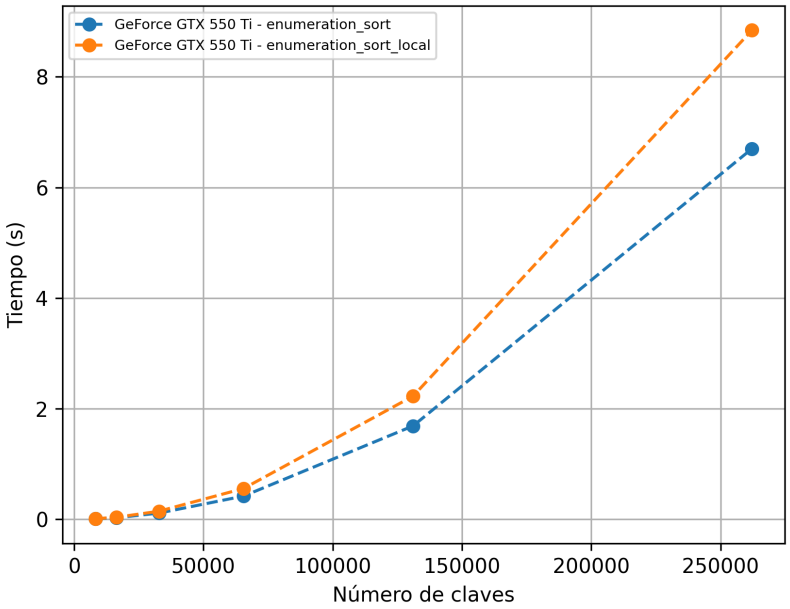

Figura 4.3: Comparación de Enumeration sort global vs local en GTX 550 Ti.

4.3. Enumeration sort, Merge sort y Radix sort

En la figura 4.4 se muestra el rendimiento de todos los algoritmos implementados ejecutados en la GPU GTX 550 Ti. Como se observa, el rendimiento de ambas versiones de Enumeration sort es mucho peor que el de las versiones de Merge sort y Radix sort.

Para visualizar mejor la diferencia entre Merge sort y Radix sort se muestra la figura 4.5. En ella observamos que Radix sort rinde peor para vectores de pequeño tamaño, pero mejor para vectores de un tamaño mayor de, aproximadamente, 40000 claves. Esto se debe a que la versión implementada del algoritmo Radix sort requiere de la creación de varios buffers para realizar la ordenación.

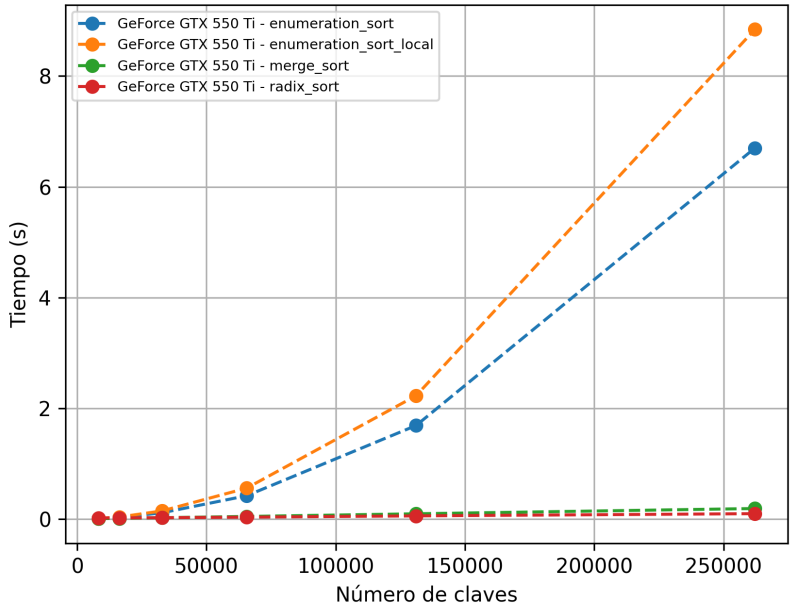

Figura 4.4: Comparación de algoritmos en GTX 550 Ti

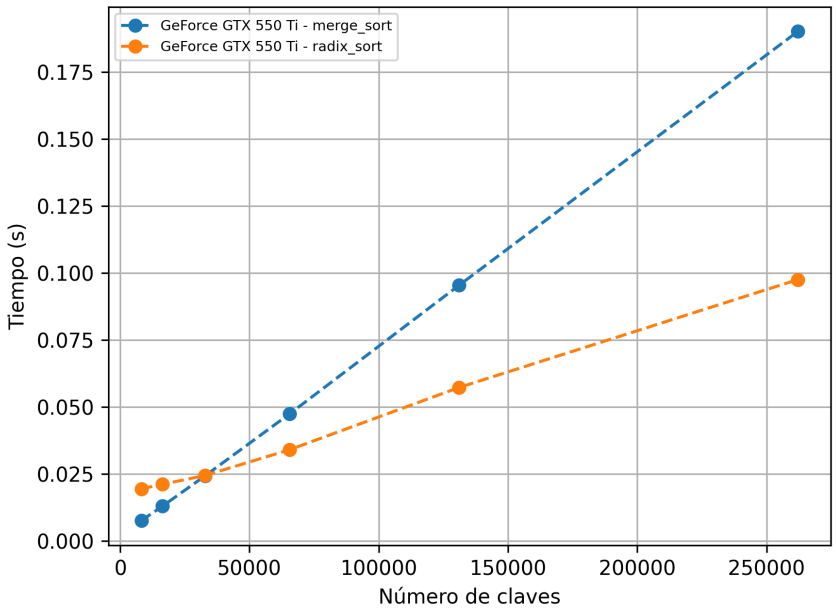

Figura 4.5: Rendimiento de Merge sort y Radix sort en GTX 550 Ti

4.4. Enumeration sort CPU y GPU

En la figura 4.6 se observa como varía el tiempo de ejecución en función del porcentaje de carga de trabajo que se ejecuta en la CPU. Estos datos son los generados y utilizados por el algoritmo descrito en el capítulo de desarrollo que se encarga de decidir cual es la distribución óptima. En mi sistema de sobremesa, como mi CPU y mi GPU tienen un rendimiento similar al ejecutar Enumeration sort, se forma un valle en el centro de la gráfica. Como se observa, la CPU rinde un poco mejor, por eso el porcentaje óptimo de trabajo en la CPU es el 60 %, lo que significa que en la GPU se ejecutaría el 40 % restante.

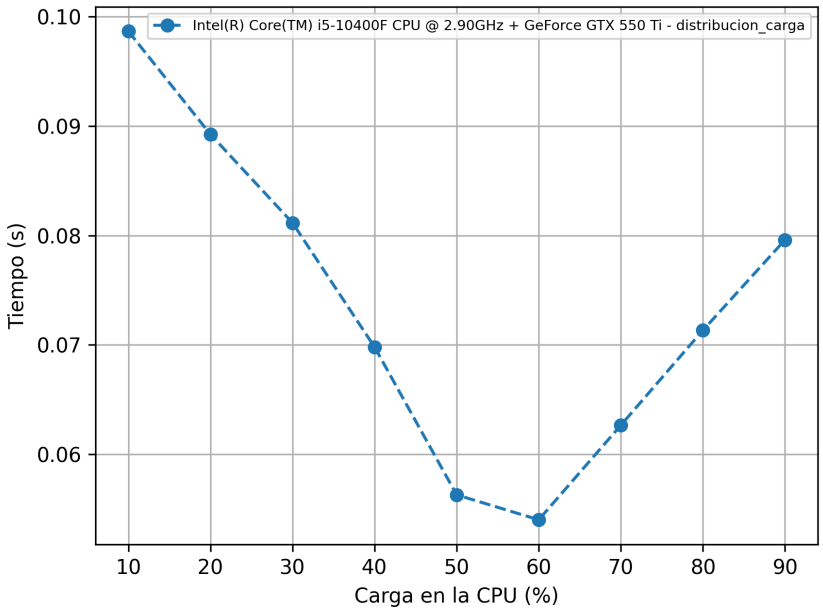

Figura 4.6: Enumeration sort CPU y GPU en función de la distribución de la carga

En la figura 4.7 se muestra el rendimiento de la versión de Enumeration sort que se ejecuta en un único dispositivo ejecutada en la CPU y la GPU por separado y el rendimiento de la versión de Enumeration sort que utiliza ambos dispositivos de cómputo utilizando como parámetro para distribuir la carga el 60 % elegido anteriormente. Como se espera, el rendimiento de esta última versión es superior a la otra. Se logra casi el doble de rendimiento frente a la versión que se ejecuta únicamente en la CPU o la GPU.

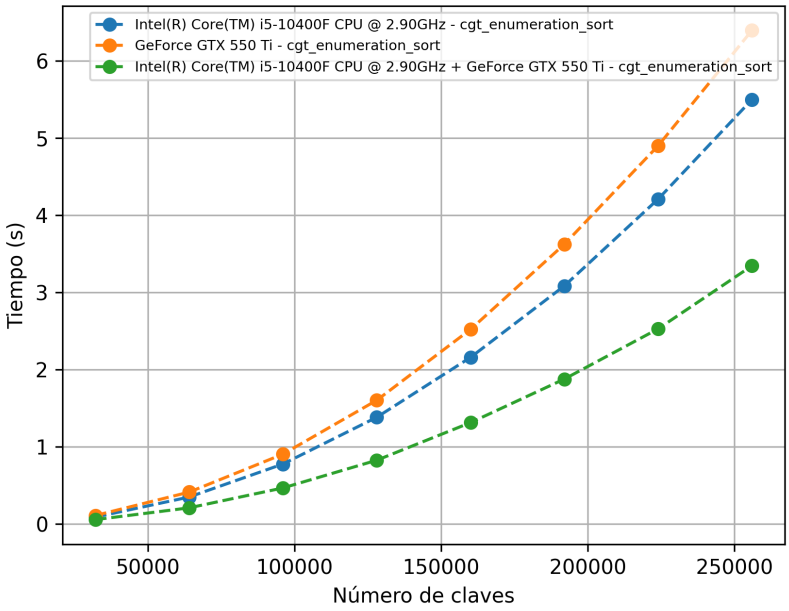

Figura 4.7: Enumeration sort CPU y GPU

5

RESULTADOS

En este capítulo se muestran las diferencias de rendimiento entre distintas GPUs para los mismos algoritmos. Se realizan comparaciones entre el rendimiento de los algoritmos en CPU y GPU, y se compara el algoritmo Enumeration sort para CPU y GPU en distintos sistemas.

5.1. Comparación de algoritmos en distintas GPUs

En las siguientes gráficas se muestra el rendimiento de los distintos algoritmos en distintas GPUs. A parte de en la GTX 550 Ti, de la que ya se ha hablado, los algoritmos se han probado en una GTX 1060 y una RTX 2060 SUPER, ambas son GPUs mucho más modernas que la GTX 550 Ti. La GTX 1060 salió al mercado en 2016 con un precio aproximado de 280€ y la RTX 2060 SUPER salió al mercado en 2019 con un precio aproximado de 420€.

Las arquitecturas de las GPUs GTX 1060 y RTX 2060 SUPER son diferentes que la de la GTX 550 Ti, utilizan arquitecturas que evolucionaron de la arquitectura Fermi. Comenzando con Enumeration sort, como cabe de esperar y se observa en la figura 5.1, la GTX 1060 y la RTX 2060 SUPER son mucho mejores que la GTX 550 Ti.

Para observar mejor la diferencia de rendimiento entre la GTX 1060 y la RTX 2060 SUPER podemos observar la figura 5.2. Al contrario que en la GTX 550 Ti, las versiones de Enumeration sort que utilizan memoria local rinden mejor que la versión que utiliza memoria global en estas dos GPUs. Esto puede deberse a que las gráficas más modernas poseen memorias VRAM más rápidas, lo que significa que los datos tardarían menos tiempo en moverse desde las memorias VRAM hasta la memoria local y por ello merece la pena utilizar la memoria local.

Con respecto a Merge sort, en la figura 5.3 se observa que la diferencia entre la GPU GTX 550 Ti y las otras dos más modernas no es tan grande como con el algoritmo Enumeration sort. Esto se debe a que la versión de Merge sort implementada no aprovecha el paralelismo tanto como la versión implementada del algoritmo Enumeration sort. Las gráficas más modernas poseen más SMs y cada uno de estos SMs posee más unidades de ejecución de tipo SIMD.

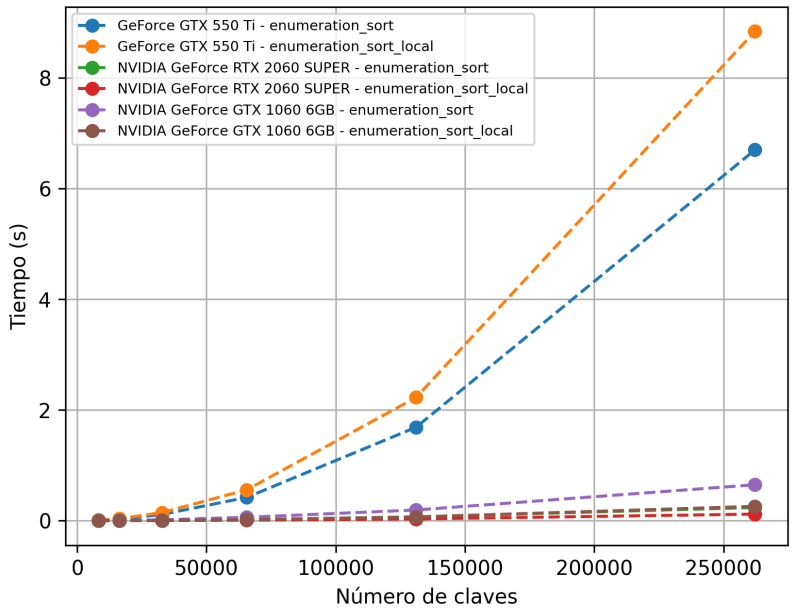

Figura 5.1: Enumeration sort en distintas GPUs

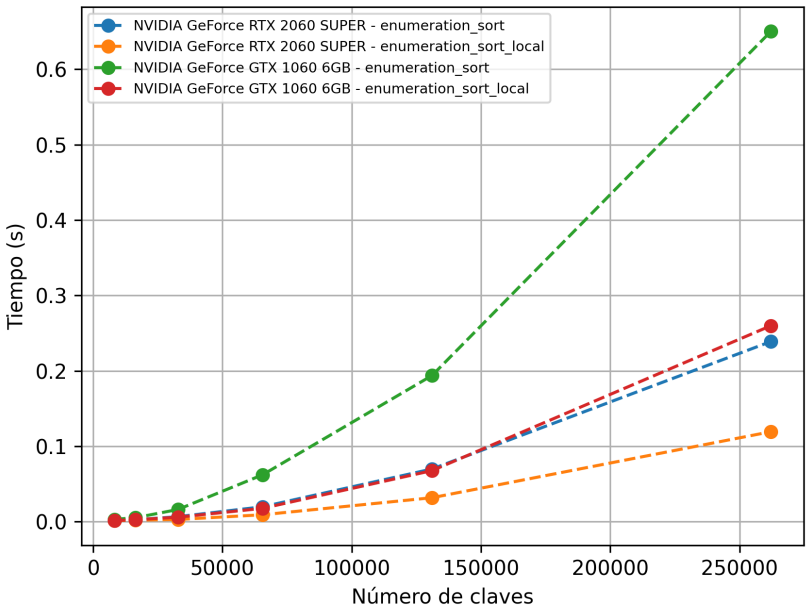

Figura 5.2: Enumeration sort en GTX 1060 y RTX 2060 SUPER

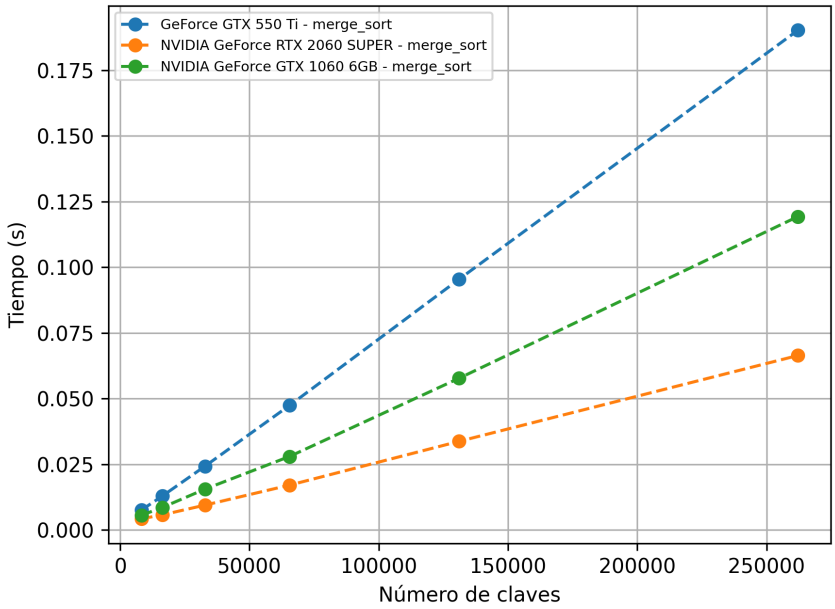

Figura 5.3: Merge sort en distintas GPUs

En la comparación de Radix sort en la figura 5.4 se observa aún menos diferencia de rendimiento entre la GTX 550 Ti y las otras dos GPUs más modernas, pero especialmente con la GTX 1060. Esto se debe a que la versión de Radix sort implementada es la que más se apoya en el código del anfitrión para ordenar el vector de entrada. Por tanto, la CPU tiene bastante impacto en este algoritmo.

5.2. Comparación de algoritmos entre CPU y GPU

En esta sección se compara el rendimiento de cada algoritmo entre la CPU y la GPU de mi sistema. En la figura 5.5 se observa que la CPU obtiene un mejor rendimiento con Enumeration sort, aunque bastante similar, que la GPU. En 5.6 se observa que la versión de Merge sort se adapta mucho mejor a la CPU que a la GPU. Como ya se ha comentado antes, la versión de este algoritmo no aprovecha al máximo la capacidad de paralelización de las GPUs. En la figura 5.7 se observa que Radix sort, al contrario que Merge sort, sí aprovecha la capacidad de paralelización de las GPUs y por eso el rendimiento que se obtiene en la GPU es mayor que en la CPU.

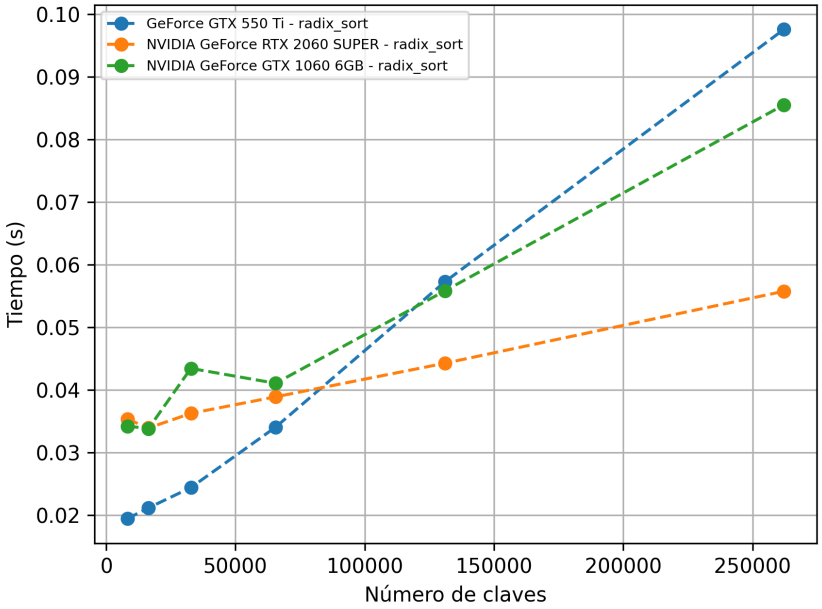

Figura 5.4: Radix sort en distintas GPUs

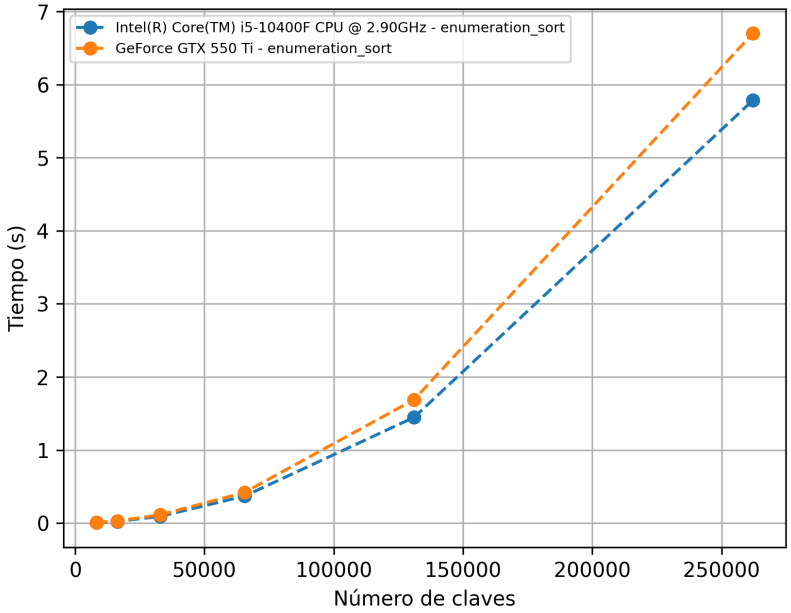

Figura 5.5: Enumeration sort en CPU y GPU

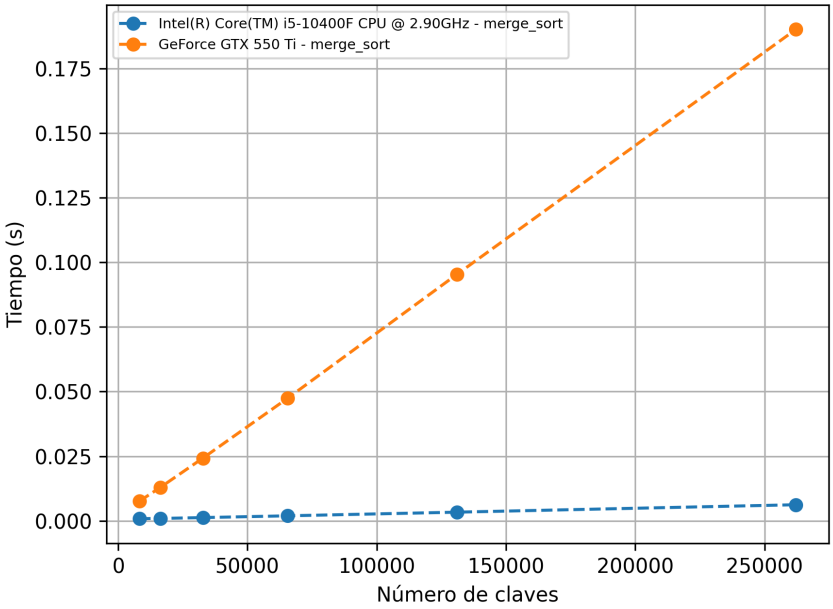

Figura 5.6: Merge sort en CPU y GPU

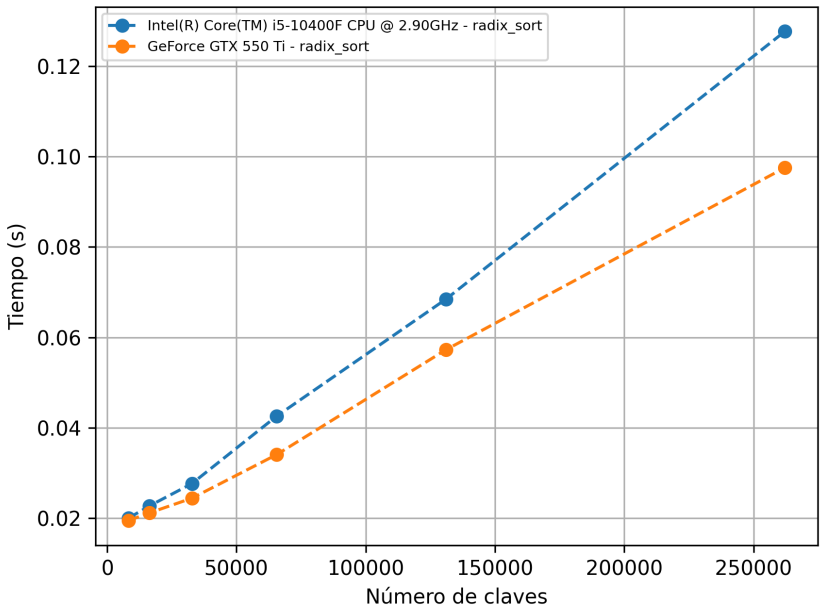

Figura 5.7: Radix sort en CPU y GPU

Si comparamos el rendimiento del mejor algoritmo en CPU, es decir, Merge sort, mostrado en la figura 5.6, con el rendimiento del mejor algoritmo en GPU, Radix sort, mostrado en la figura 5.7, se observa que, teniendo en cuenta únicamente los algoritmos implementados, la CPU es muy superior a la GPU para resolver problemas de ordenación. Se debe recordar que la comparación entre estos dispositivos no es justa pues entre ellos existe casi 10 años de diferencia y además es muy posible que los algoritmos utilizados puedan ser mucho más optimizados para GPU.

5.3. Enumeration sort CPU y GPU en distintos sistemas

En esta sección se comparan los datos generados y utilizados por el algoritmo que decide la distribución óptima de la carga de trabajo para la versión de Enumeration sort que se ejecuta en dos dispositivos de cómputo distintos, en este caso éstos son la CPU y GPU de distintos sistemas. Los sistemas en los que se ha realizado la prueba son tres:

- El sistema de la GTX 550 Ti y el Intel i5-10400F utilizado anteriormente.
- El sistema de la RTX 2060 SUPER utilizada anteriormente que contiene un procesador i7-4790K.
- El sistema de la GTX 1060 que contiene un procesador i7-9700K.

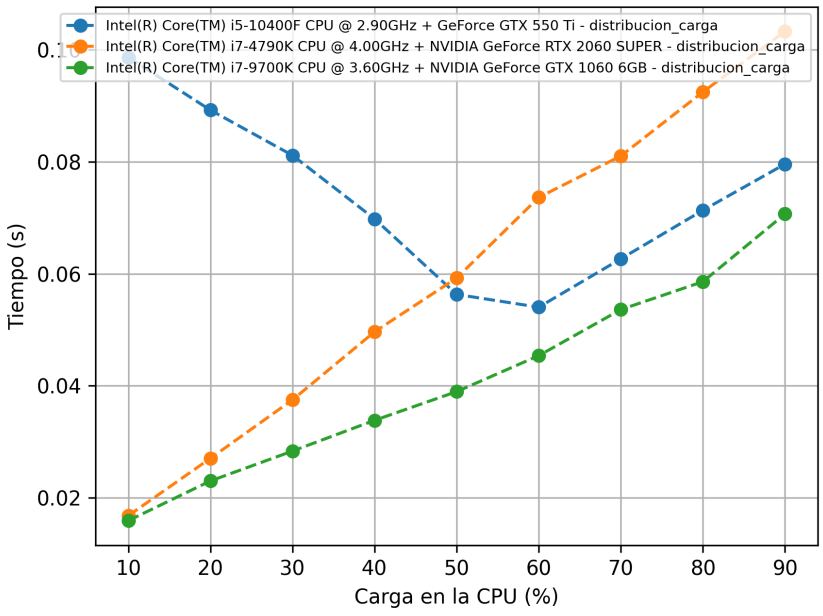

Figura 5.8: Enumeration sort CPU y GPU en distintos sistemas

Los resultados de las pruebas se muestran en la figura 5.8. Como se observa, el único sistema en que merece la pena utilizar la CPU para procesar parte del trabajo, es el sistema formado por el Intel i5-10400F y la GTX 550 Ti.

Utilizar la CPU para procesar parte de la carga de trabajo en los sistemas de la GTX 1060 y la RTX 2060 SUPER no es beneficioso puesto que las GPUs son capaces de lograr un mayor rendimiento en comparación con el rendimiento de sus respectivas CPUs. Esto es de esperar en el caso del sistema de la RTX 2060 SUPER, pues ésta es mucho más moderna que la CPU con la que trabaja. Y en el caso del sistema con la GTX 1060, observamos un claro ejemplo del potencial que están alcanzando las GPUs para resolver problemas que se adaptan bien a su arquitectura. La GTX 1060 se lanzó al mercado en 2016 con un precio aproximado de 300 € y la CPU Intel i7-9700K se lanzo en 2018 con un precio aproximado de 400 €, y es la GPU la que rinde mejor con el algoritmo empleado.

6

Conclusiones y trabajo futuro

Como se ha observado, aunque el código desarrollado con el framework OpenCL pueda ser ejecutado en muchos dispositivos de cómputo, el rendimiento obtenido en cada dispositivo puede ser muy diferente. Unas aplicaciones funcionan mejor que otras en un mismo dispositivo y puede suceder lo contrario en otros dispositivos como se ha visto en la comparación de Merge sort y Radix sort en la CPU y la GPU.

Puesto que el rendimiento depende en gran parte de como se lleva a cabo la adaptación de las soluciones a cada dispositivo o arquitectura, para crear aplicaciones óptimas se deben conocer los detalles de cada arquitectura. Por ejemplo, se ha observado que la versión de Enumeration sort que utilizaba memoria local funcionaba peor que la versión que utilizaba memoria global en la GPU GTX 550 Ti y sucedía lo contrario en las GPUs GTX 1060 y RTX 2060 SUPER.

Por otro lado, en el desarrollo de aplicaciones para sistemas heterogéneos, si se prioriza desarrollar una aplicación multiplataforma frente a una aplicación con muy buen rendimiento, se ha confirmado y probado que el framework OpenCL es una muy buena opción por su simplicidad y abstracción.

Como líneas de trabajo futuro se proponen algunas ideas y cuestiones que surgieron durante el desarrollo del trabajo. En primer lugar, sería interesante estudiar las arquitecturas más modernas de GPUs con el objetivo de analizar las diferencias entre éstas y las más antiguas como Fermi. Por otro lado, se podría realizar una comparación del rendimiento que se puede alcanzar con las versiones actuales de los frameworks CUDA, OpenCL y SYCL ya que el trabajo referenciado en este trabajo, donde se compara CUDA y OpenCL, es del año 2011.

También se propone seguir trabajando e investigando sobre mecanismos que faciliten la creación de aplicaciones OpenCL donde la carga de trabajo se reparte entre dispositivos correspondientes a distintas plataformas. Y por último, se plantea estudiar diferentes alternativas para optimizar y realizar el reparto de la carga de trabajo en dichas aplicaciones.

Bibliografía

[1] B. Gaster, *Heterogeneous computing with OpenCL*. Amsterdam ; Boston: Elsevier/MK, 2nd ed. ed., 2013.

[2] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[3] D. P. Singh, D. P. Singh, I. Joshi, I. Joshi, J. Choudhary, and J. Choudhary, "Survey of gpu based sorting algorithms," *International journal of parallel programming*, vol. 46, no. 6, pp. 1017–1034, 2018.

[4] D. I. Arkhipov, D. Wu, K. Li, and A. C. Regan, "Sorting with gpus: A survey," 2017.

[5] P. N. Glaskowsky, "Nvidia's fermi: The first complete gpu computing architecture." `https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf`.

[6] "Cuda." `https://developer.nvidia.com/cuda-zone`.

[7] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," 2011.

[8] K. O. W. Group, "The opencl specification 2.2." `https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf`.

[9] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[10] "Documentación pyopencl." `https://documen.tician.de/pyopencl/`.

[11] "Sycl." `https://www.khronos.org/sycl/`.

[12] E. Manca, A. Manconi, A. Orro, G. Armano, and L. Milanesi, "Cuda-quicksort: An improved gpu-based implementation of quicksort," *Concurr. Comput.: Pract. Exper.*, vol. 28, p. 21–43, Jan. 2016.

[13] "Tinydb." `https://tinydb.readthedocs.io/en/latest/`.

[14] "Pyinstaller." `https://www.pyinstaller.org/`.